Automatic Synthesis of Safety-Related Software[*] — Short Paper —

Johann Schumann
RIACS / NASA Ames
email:`schumann@email.arc.nasa.gov`

Abstract

For specific domains (e.g., data analysis, planning and scheduling, or state estimation), automated program synthesis systems have been developed which are capable of producing hundreds of lines of non-trivial code. However, the potential applicability of an automatic program synthesis system does not only depend on size and quality of the generated code, but also its ability to be integrated into the overall software process. Therefore, the generation of executable code alone is not enough. In this paper, we will describe three techniques which enhance the capabilities of a synthesis tool with respect to generation of explanations, certificates, and simulation data. The synthesis system encodes enough domain knowledge, such that the appropriate information can directly be extracted during the synthesis process.

ExplainIt! is a component for the AMPHION/NAV system (synthesis of state estimation software) which automatically generates and displays explanations for each piece of the synthesized code, thus effectively achieving traceability between code and specification.

For safety-relevant applications, software must undergo a rigorous certification process where it must be demonstrated that certain safety policies are not violated. Traditional formal verification approaches (e.g., with Hoare-style rules) are impractical, because they require large amounts of manual code annotations. In this paper, we discuss an extension of the AUTOBAYES system (synthesis of data analysis programs) for the automatic generation of code annotations which can be handled by a verification condition generator and an automated theorem prover. Speed of this approach compares favorably with commercial static analysis tools (e.g., PolySpace).

Finally, we discuss a module of AUTOBAYES which synthesizes code for the generation of artificial data for simulation, experimentation, and testing purposes.

Introduction

Over the recent years, size and complexity of software in safety-related areas has grown tremendously. A major reason for this is that functionality which has been traditionally realized by hardware is now implemented as a program on a general-purpose processor, thus reducing production costs and increasing functionality. Typical application areas range from avionics, process control (e.g., for chemical or nuclear plants) to car industry. However, the production of reliable, high-quality code for safety-related applications is far from easy. In particular, modern, highly iterative software lifecycles (e.g., spiral or use-case based processes) are major cost drivers, because for each iteration, substantial testing, documentation, and certification efforts are necessary. For example, flight-critical software (e.g., for position estimation or control of an aircraft) requires rigorous certification by an independent certification authority (e.g., the FAA). This time-consuming, highly manual process which is defined in standard documents (e.g., DO-178B) prescribes the required testing, documentation, and engineering efforts to guarantee traceability between specification and the executable binary.

An approach which could facilitate the production of such pieces of software is automated program synthesis. Given a high-level specification, an automated program synthesis tool generates executable code which implements the specification. Because rigorous formal logic underlies this approach, the synthesized code is often considered to be "correct-by-construction".

Deduction-based program synthesis is around for a long time, and several synthesis systems (e.g., Amphion (Stickel *et al.* 1994), KIDS (Smith 1990), or Planware (Burstein & Smith 1996)) have been developed over the years and it seems that in certain (albeit small) domains such systems are capable of producing reasonably good code. However, the usability of such systems in the area of safety-related domains is still rather limited. In fact, they share many severe limitations with state-of-the-art code generators for traditional modeling systems (e.g., MatrixX (MatrixX 2001), ControlShell (ControllShell 2001)). As discussed above, production of a piece of code is not enough. Rather, a code-producing system needs to synthesize the following artifacts:

- well documented, human-understandable code. Only if a piece of software can be easily understood, manual modifications can be applied or it can be subject to (successful) code reviews.
- traceability information between code and specification

[*]This paper discusses work done in several synthesis projects at the Automated Software Engineering group (Guillaume Brat, Bernd Fischer, Mike Lowry, John Penix, Tom Pressburger, Phil Oh, Grigore Rosu, Mahadevan Subramaniam, Jeffrey van Baalen, Jonathan Whittle).

such that all pieces of the code can be related to their origin in the specification.

- support for simulation, animation, and testing. A successful synthesis system needs to be able to produce artificial data which conform to the given specification. State-of-the-art modeling tools (e.g., Simulink/MatLab, ControllShell) are already pretty advanced with that respect.

- support for certification (e.g., providing annotations or even proofs).

In this paper, we demonstrate that a program synthesis system encodes enough domain knowledge to support the requirements listed above. We will discuss three extensions to a program synthesis architecture which, in addition to producing executable code, generate detailed documentation/explanations, certificates for the synthesized code with respect to a given safety policy, and test/simulation data, respectively.

The work, described in this paper, is ongoing work. Therefore, these extensions have not been developed for one single program synthesis system, but rather for two tools, namely AMPHION/NAV and AUTOBAYES. AMPHION/NAV (Whittle *et al.* 2001,-Schumann & Robinson 2001) is a tool based on the Amphion system (Stickel *et al.* 1994) which is capable of automatically synthesizing C/C++ code for state-estimation and navigation of aircraft or spacecraft. The domain of AUTOBAYES (Fischer & Schumann 2001,-Fischer, Schumann, & Pressburger 2000) is data analysis, using the approach of Bayesian networks. This tool can be used for scientific data analysis (e.g., clustering or classification problems), but it also can synthesize code to model sensors and sensor failures. Both systems are aim toward applications where safety is important, for example, state-estimation of Mars rovers or (on-board) scientific data analysis.

Architecture of an Extended Synthesis System

Figure 1 shows the system architecture of a modern, extended program synthesis system. Given a specification, the synthesis system produces executable code. For this core task, domain knowledge in form of a domain theory is used to guide the synthesis process. The underlying principle of the synthesis engine is of no great importance for the discussion in this paper. For example, the AMPHION/NAV system is based upon deduction-based synthesis (using the first-order theorem prover SNARK), whereas AUTOBAYES uses schema-guided synthesis. However, all these systems have in common that they rely on a substantial body of encoded domain knowledge. This domain knowledge, combined with information on how the program was assembled (e.g., a proof) can be used to extend the synthesis system to produce commented code, design documents, test data, and support for rigorous certification. These extensions will be described in the following sections.

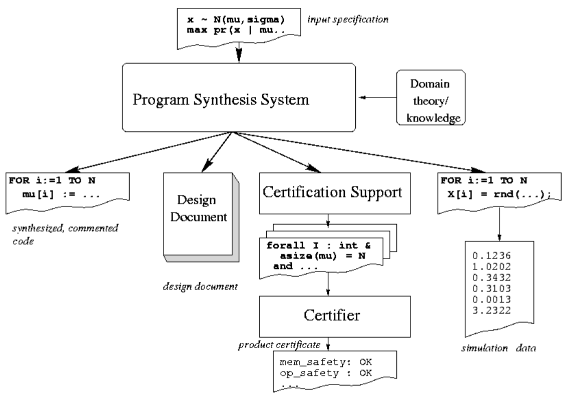


Figure 1: System Architecture for an Extended Program Synthesis System

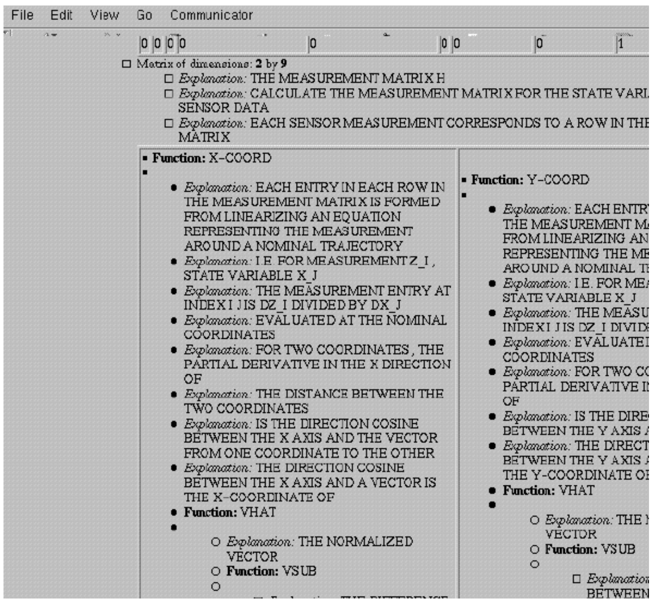
Explaining Synthesized Code

In the AMPHION/NAV system, most axioms in the domain theory[1] are given as a set of first-order equations. These equations relate the various objects on different abstraction levels. Due to the synthesis process (deductive synthesis) and additional program transformation steps, it is nearly impossible to tell which parts of the synthesized code corresponds to which part of the specification, or why the code is structured in a specific way. In a safety-related application environment, traceability between specification and code is of major importance. During manual development of such software, considerable effort is spent on writing detailed documentation on all aspects of the code.

Here, deductive program synthesis can help, because all information relating specification, code, and domain theory is available in the *proof* produced by the automated theorem prover. The proof, containing hundreds of inference steps is converted in such a way that it relates the input specification with the final product (C/C++ code). The explanation thus can be seen as a description of the program design "from first principle". AMPHION/NAV contains the subsystem "ExplainIt!" which produces explanations for the synthesis task (for details see (Whittle *et al.* 2001,-Schumann & Robinson 2001)). Each axiom of the domain theory is annotated by explanation templates, consisting of plain text and (logical) variables. Whenever an axiom is used for the proof, the variables in the templates are instantiated. In order to find the entire explanation, a set of explanation equalities (van Baalen *et al.* 1998) is generated which is used to compose the corresponding explanation templates.

Human readability and understandability of such an explanation is extremely important. However, the target audience is not a logically trained synthesis person, but a domain expert/engineer. This means that not only all evidence of

[1]The domain theory for AMPHION/NAV is built on top of the domain theory of the AMPHION system (Stickel *et al.* 1994) on geometric relationships, coordinate systems, and celestial mechanics.



0 0 0 0 0 0 0 0 1

☐ Matrix of dimensions: **2** by **9**

- ☐ *Explanation:* THE MEASUREMENT MATRIX H
- ☐ *Explanation:* CALCULATE THE MEASUREMENT MATRIX FOR THE STATE VARI SENSOR DATA
- ☐ *Explanation:* EACH SENSOR MEASUREMENT CORRESPONDS TO A ROW IN THE MATRIX

▪ **Function:** X-COORD

▪

- *Explanation:* EACH ENTRY IN EACH ROW IN THE MEASUREMENT MATRIX IS FORMED FROM LINEARIZING AN EQUATION REPRESENTING THE MEASUREMENT AROUND A NOMINAL TRAJECTORY
- *Explanation:* I.E. FOR MEASUREMENT Z_I , STATE VARIABLE X_J
- *Explanation:* THE MEASUREMENT ENTRY AT INDEX I J IS DZ_I DIVIDED BY DX_J
- *Explanation:* EVALUATED AT THE NOMINAL COORDINATES
- *Explanation:* FOR TWO COORDINATES , THE PARTIAL DERIVATIVE IN THE X DIRECTION OF
- *Explanation:* THE DISTANCE BETWEEN THE TWO COORDINATES
- *Explanation:* IS THE DIRECTION COSINE BETWEEN THE X AXIS AND THE VECTOR FROM ONE COORDINATE TO THE OTHER
- *Explanation:* THE DIRECTION COSINE BETWEEN THE X AXIS AND A VECTOR IS THE X-COORDINATE OF
- **Function:** VHAT
-
    - ○ *Explanation:* THE NORMALIZED VECTOR
    - ○ **Function:** VSUB
    - ○

▪ **Function:** Y-COORD

▪

- *Explanation:* EACH ENTR THE MEASUREMENT M FROM LINEARIZING AN REPRESENTING THE ME AROUND A NOMINAL T
- *Explanation:* I.E. FOR ME STATE VARIABLE X_J
- *Explanation:* THE MEASU INDEX I J IS DZ_I DIVID
- *Explanation:* EVALUATE COORDINATES
- *Explanation:* FOR TWO C PARTIAL DERIVATIVE I OF
- *Explanation:* IS THE DIRE BETWEEN THE Y AXIS
- *Explanation:* THE DIRECT BETWEEN THE Y AXIS THE Y-COORDINATE O
- **Function:** VHAT
-
    - ○ *Explanation:* THE VECTOR
    - ○ **Function:** VSUB
    - ○
        - ☐ *Explanatio* BETWEEN

Figure 2: Screen dump of a part of the explanation document

low-level deduction needs to be hidden from the user. Furthermore, the representation of data should use form and vocabulary of the domain. In the domain of AMPHION/NAV, the commonly used data structures are vectors and matrices (as opposed to lists and lists-of-lists in AMPHION/NAV's internal representation). Thus, explanation of a matrix is best represented in a tabular form, as shown in the screen-dump in Figure 2. It shows a part of the explanation for a matrix ("measurement matrix H") which relates the measurements with the current position estimate. Each cell of the table corresponds to a single entry in the matrix. This HTML document is produced from the internal XML representation which is generated by "ExplainIt!". Translation/formatting is done with XSLT. Hyperlinked HTML documents have the advantage that all statements of the synthesized code can be linked to their explanations. Thus, a simple click on a statement immediately produces the related documentation. Using XML as a flexible internal document format enables us to also generate printed PDF documentation in a standardized form.

Certifying Synthesized Code

Code certification is a lightweight approach (as opposed to e.g., full functional verification) to demonstrate software quality on a formal level. Its basic idea is to produce formal proofs demonstrating that the code satisfies certain quality properties (e.g., memory or operator safety). These proofs can be seen as certificates (for the produced code) which can be checked independently by a simple proof checker. Since code certification uses the same underlying technology as Hoare-style program verification, it also requires many detailed annotations (e.g., loop invariants) to make the proofs possible. However, manually adding these annotations to the code is an extremely time-consuming and error-prone task..

In a certification extension of AUTOBAYES, we address this problem (Whalen, Schumann, & Fischer 2002). AUTOBAYES contains sufficient high-level domain knowledge to generate the required detailed annotations. Because all constraints and information on design decisions is available during synthesis time, detailed and powerful local annotations can be generated easily by AUTOBAYES. A separate propagation algorithm distributes the annotations to all places in the code where they are valid. When annotations were generated by AUTOBAYES, the original 380 lines of commented code grew to more than 2100 lines of code with annotations. This is a clear indication that writing manual annotations are infeasible.

From this annotated code, a general-purpose verification condition generator (in our case MOPS (Kaiser, Fischer, & Struckmann 2000)) produces a set of proof obligations in first-order logic. The obligations are then processed by the automated theorem prover E-SETHEO (CASC 2001).

In (Whalen, Schumann, & Fischer 2002) we have demonstrated our approach by certifying operator safety and memory safety for a generated iterative data classification program (≈ 380 lines of documented C++ code) without manual annotation of the code. For this example, a total of 69 proof tasks have been generated. E-SETHEO could solve 65 automatically with a run-time limit of 60 seconds on a 1000 MHz SunBlade workstation. Most of the tasks could be solved in about one second, but several tasks took up to 40 seconds (average time: 6.3 seconds). The remaining four proof tasks currently require some manual preprocessing which will be automated in future versions. A comparison with the state-of-the-art commercial static analysis tool PolySpace (PolySpace 2002) showed that our approach could reach a better coverage with a substantially shorter runtime.

Generation of Simulation and Test Data

Testing and simulation plays a vital role in most software development processes. Whereas testing aims at showing that the piece of code works correctly, simulation is often used to demonstrate how the code works and to assess its quality and performance. Therefore, the availability of simulation and test-data is of great importance. To set up a simulation environment manually, however, is usually a very time consuming and error prone task. This is especially true when the requirements specifications are modified in a rapid succession (e.g., in an iterative life cycle).

With program synthesis, the development of a simulation environment can be very straightforward; we synthesize a program from our given specification which generates test data. The advantages are obvious: we already have a specification, and most of the synthesizer's infra-structure (e.g., symbolic handling, code generation) can be used as is for this task. For AUTOBAYES, we have developed a tool component which can synthesize a program to generate randomized data according to the given specification. This data generator could be implemented in less than 200 lines of Prolog code on top of the AUTOBAYES system.

Conclusions

In this paper, we have briefly described three extensions to bare-bones program synthesis technology which can increase usability of a synthesis tool in safety-related application areas. In the AMPHION/NAV system, a detailed explanation is generated fully automatically and presented in a way suitable for the domain engineer. It fully hides the underlying logic and reasoning system used to synthesize the program. The proof steps is converted in such a way that it relates the input specification with the final product, thus opening up an entirely new level of traceability between specification and source code.

Explanation and documentation is only one aspect. Current practice of certification of safety-critical code requires huge testing effort and lengthy manual code reviews. Automatic certification of synthesized code has the potential to substantially facilitate and accelerate certification. In combination with techniques from proof-carrying code (Necula & Lee 1998), dynamic certification of field-loadable software can be addressed. Here again, we benefit from the fact, that the synthesis system encodes enough domain knowledge such that the required Hoare-style annotations can be made automatically. Last, but not least, trying out synthesized code during simulation runs is an important feature for a practical usable system. The test data generator provides immediate feed-back on the specification (does it make sense or are there some obvious bugs?) and helps to navigate through the design space.

All those features form essential ingredients of a modern program synthesis system if it should have a chance to be used in practice. Bare-bones synthesis power does not help here, it only leads to repeating the same mistakes as have been made with automated theorem provers, which are usually restricted "more by general usability than by raw deductive power"[2].

References

Burstein, M. B., and Smith, D. 1996. ITAS: A Portable Interactive Transportation Scheduling Tool Using a Search Engine Generated from Formal Specifications. In *Proceedings of the 3rd International Conference on AI Planning Systems (AIPS-96)*, 35–44. AAAI Press.

CASC-JC, 2001. The CASC-JC theorem proving competition. URL: `http://www.cs.miams.edu/~tptp/CASC/JC`.

Controlshell. 2001. RTI Real-Time Innovations. `http://www.rti.com`.

Fischer, B., and Schumann, J. 2001. AutoBayes: A system for generating data analysis programs from statistical models. Submitted for publication. Preprint available at `http://ase.arc.nasa.gov/people/...` `fischer/papers.html`.

Fischer, B.; Schumann, J.; and Pressburger, T. 2000. Generating data analysis programs from statistical models (position paper). In Taha, W., ed., *Proc. Intl. Workshop Semantics Applications, and Implementation of Program Generation*, volume 1924 of *Lect. Notes Comp. Sci.*, 212–229. Montreal, Canada: Springer.

Kaiser, T.; Fischer, B.; and Struckmann, W. 2000. Mops: Verifying Modula-2 programs specified in VDM-SL. In *Proc. 4th Workshop Tools for System Design and Verification*, 163–167.

MatrixX: AutoCode Product Overview. ISI. URL: `http://www.isi.com`.

Necula, G. C., and Lee, P. 1998. Efficient representation and validation of logical proofs. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, 93–104. IEEE Computer Society Press.

PolySpace technologies. URL: `http://www.polyspace.com`.

Schumann, J., and Robinson, P. 2001. [] or success is not enough: Current technology and future directions in proof presentation. In *Future Trends in Automated Deduction (during IJCAR 2001)*.

Smith, D. R. 1990. KIDS: A Semiautomatic Program Development System. *IEEE Trans. on Software Engineering* 16(9):1024–1043.

Stickel, M.; Waldinger, R.; Lowry, M.; Pressburger, T.; and Underwood, I. 1994. Deductive composition of astronomical software from subroutine libraries. In Bundy, A., ed., *Proc. 12th International Conference Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*, 341–355. Springer.

van Baalen, J.; Robinson, P.; Lowry, M.; and Pressburger, T. 1998. Explaining synthesized software. In *Thirteenth International Conference on Automated Software Engineering*, 240–248. IEEE Computer Society Press.

Whalen, M.; Schumann, J.; and Fischer, B. 2002. Synthesizing certified code. In *Proc. ICSE 2002*. (submitted).

Whittle, J.; van Baalen, J.; Schumann, J.; Robinson, P.; Pressburger, T.; Penix, J.; Oh, P.; Lowry, M.; and Brat, G. 2001. Amphion/NAV: Deductive Synthesis of State Estimation (short paper). In *Proceedings of the 16th Automated Software Engineering Conference 2001 (ASE 2001)*. IEEE.

[2]M. Kaufmann in his invited talk during CADE 15, 1998.